INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

NOMBRES:

PADILLA MATIAS CRISTIAN MICHEL

SAUCILLO GONZÁLEZ JESSE OBED

GRUPO:

4BM1

TRABAJO:

Practica 2

"Búsqueda no informada"

MATERIA:

Fundamentos de Inteligencia Artificial

FECHA

25 de septiembre del 2023

PRACTICA 1. AGENTES REACTIVOS

RESUMEN

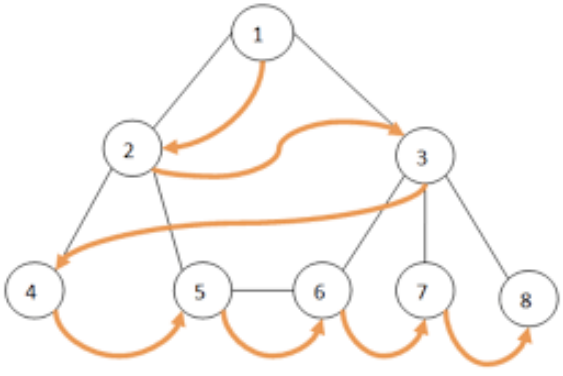
En esta práctica se realizo la creación de una aplicación de un 15-puzzle; un tablero con una imagen segmentada en 15 partes y 1 espacio en blanco de manera desordenada, donde la meta será intercambiar el espacio en blanco por cada segmento fronterizo hasta dar con el orden correcto. Dentro de esta aplicación se desarrollaron los algoritmos de búsqueda primero en anchura y búsqueda primero en profundidad, estos pueden ser seleccionados por el usuario para resolver el 15-puzzle.

INTRODUCCIÓN

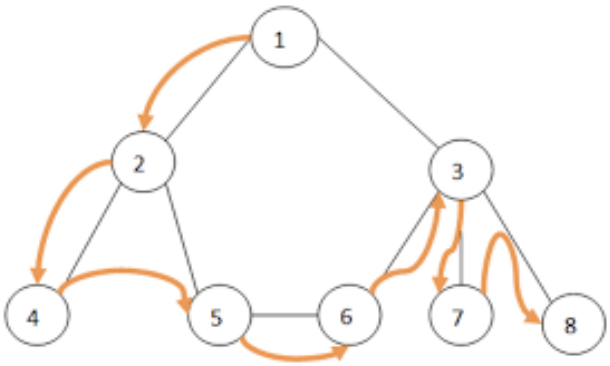
La búsqueda no informada, a veces llamada búsqueda ciega o búsqueda a ciegas, es un enfoque de búsqueda en la inteligencia artificial donde el algoritmo no tiene información previa sobre la ubicación o la estructura del objetivo que está tratando de encontrar. En otras palabras, no utiliza ninguna heurística o conocimiento adicional para guiar la búsqueda hacia la solución. En lugar de eso, se basa únicamente en la exploración sistemática del espacio de búsqueda, existen 2 estrategias de búsqueda no informada: búsqueda primero en anchura y búsqueda primero en profundidad.

Búsqueda primero en anchura (BFS - Breadth First Search)

La estrategia de búsqueda en anchura comienza por explorar el nodo raíz y luego se expanden gradualmente los nodos sucesores, seguidos de sus sucesores, y así sucesivamente. Se exploran todos los nodos en el mismo nivel de profundidad en el árbol de búsqueda antes de pasar al siguiente nivel de profundidad.



Búsqueda primero en profundidad (DFS –Depth First Search)



En la estrategia de búsqueda en profundidad, se inicia desde el nodo raíz y se sigue una de las ramificaciones del árbol lo más lejos posible hasta encontrar el nodo deseado o alcanzar un nodo hoja, que es un nodo sin descendientes. Si se llega a un nodo hoja, se continúa la búsqueda ascendiendo hacia el ancestro más cercano que tenga nodos hijos sin explorar.

15-Puzzle

El 15-puzzle es un juego de rompecabezas que consiste en una cuadrícula de 4x4 con 15 fichas numeradas y un espacio vacío. El objetivo del juego es mover las fichas para ordenarlas en orden numérico, dejando el espacio vacío en la esquina inferior derecha. Para resolver el 15-puzzle, se pueden utilizar diferentes algoritmos de búsqueda, como la búsqueda primero en anchura (BFS) y la búsqueda primero en profundidad (DFS).

DESARROLLO

El código de la práctica se realizó en lenguaje Java. Para el desarrollo de la aplicación se modifico el proyecto “Pozole” proporcionado por el profesor. Este proyecto contiene las clases: “Pozole”, “Tablero” “State”, y “Executor”. A continuación, se explica a detalle cada clase.

Pozole

Esta es la clase principal que contiene el método main. En este método se crea una instancia de la clase Tablero y se muestra en pantalla.

```
package pozole;

/**
 *
 * @author
 */
public class Pozole
{

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        Tablero t =new Tablero();
        t.setVisible(true);
    }
}
```

Tablero

Esta es la clase principal del juego y extiende la clase JFrame de Java Swing para crear una ventana de juego. Aquí se configura la interfaz gráfica del juego, se carga la imagen del rompecabezas, se manejan eventos de botones y se inicia la resolución del rompecabezas utilizando los algoritmos de búsqueda en anchura (BFS) o búsqueda en profundidad (DFS) según la elección del usuario.

initComponents: Configura la interfaz gráfica del juego, crea y muestra botones, y maneja eventos.

```
private void initComponents()
{
    this.setTitle("15-Puzzle");
    this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

    Dimension pantalla = Toolkit.getDefaultToolkit().getScreenSize();
    int width = pantalla.width;
    int height = pantalla.height;
    this.setBounds((width-516)/2,(height-563)/2,516,563);

    JMenuBar mainMenu = new JMenuBar();
    JMenu    file = new JMenu("File");
    JMenuItem exit = new JMenuItem("Exit");

    mainMenu.add(file);
    file.add(solveB);
    file.add(solveD);
    file.add(exit);
    this.setJMenuBar(mainMenu);

    this.setLayout(null);
    this.imagePieces("imagenes/universo.jpg");
    paintPieces();
    exit.addActionListener(evt -> gestionarExit(evt));
    solveB.addActionListener(evt -> whichMethod(evt));
    solveD.addActionListener(evt -> whichMethod(evt));
```

goodBye: Cierra la aplicación.

```
private void gestionarExit(ActionEvent e)
{
    goodBye();
}
```

gestionarExit: Maneja la acción de salida de la aplicación.

```
private void gestionarExit(ActionEvent e)
{
    goodBye();
}
```

imagePieces: Divide una imagen en piezas para usar en el rompecabezas.

```
private void imagePieces(String pathName)
{
    try  // Bloque "try" por que hay una tarea de lectura de archivo
    {
        BufferedImage buffer= ImageIO.read(new File(pathName));
        BufferedImage subImage;
        int n=0;
        for(int i=0;i<4;i++)
            for(int j=0;j<4;j++)
            {
                subImage = buffer.getSubimage((j)*125, (i)*125, 125, 125);
                String k = goal.substring(n,n+1);
                puzzle.put(k, subImage); // Almacena las piezas etiquetándo
                n++;
            }
    }
    catch (Exception ex)
    {
        ex.printStackTrace(System.out);
    }
```

paintPieces: Coloca las imágenes en los botones del tablero de juego.

```
public void paintPieces()
{
    int n=0;
    for(int i=0;i<4;i++)
        for(int j=0;j<4;j++)
        {
            String  k=start.substring(n,n+1);
            BufferedImage subImage = (BufferedImage) puzzle.get(k);
            jBoard[i][j] = new JButton();
            jBoard[i][j].setBounds(j*125+1, i*125+1,125,125); // Calcula la posición del botón i,
            this.add(jBoard[i][j]);
            if(!k.equals("0"))jBoard[i][j].setIcon(new ImageIcon(subImage));
            else empty = subImage;
            n++;
        }

}
```

whichMethod: Determina si se utilizará BFS o DFS para resolver el rompecabezas.

```
private void whichMethod(ActionEvent e)
{
    if(e.getSource()==solveD) dept=true;
    solve();
}
```

solve: Inicia la resolución del rompecabezas usando BFS o DFS.

```
private void solve()
{
    boolean success = false;
    int deadEnds = 0;
    int totalNodes = 0;
    State startState = new State(start);
    State goalState   = new State(goal);
    ArrayDeque queue = new ArrayDeque();
    ArrayList<State> first = new ArrayList();
    ArrayList<State> path=null;
    solveB.setEnabled(false);
    solveD.setEnabled(false);

    first.add(startState);
    queue.add(first);

    // Ciclo de búsqueda

    boolean deepCond = false;
    dr=0;
    int m=0;
    int contador=0;
    long startTime = System.currentTimeMillis();
    while(!queue.isEmpty() && !success && !deepCond)
    {
        System.out.println("Queue vacia: "+queue.isEmpty());
        System.out.println("Estado succes: "+success);
        System.out.println("Estado deepCond: "+deepCond);
        m++;

      int validStates = 0;
      //System.out.println("Ciclo " + m);
      ArrayList<State> l = (ArrayList<State>) queue.getFirst()
      //System.out.println("Analizando Ruta de :" + l.size());
      // muestraEstados(l);
      State last = (State) l.get(l.size()-1);
      //last.show();
      ArrayList<State> next = last.nextStates();
      //System.out.println("Se encontraron " + next.size()+ " esta
      totalNodes+=next.size();

      queue.removeFirst(); // Se elimina el primer camino de la es

      for(State ns: next)
      {
          if(!repetido(l,ns)) // Se escribió un método propio para
          {
              validStates++;
              ArrayList<State> nl = (ArrayList<State>) l.clone();
              if(ns.goalFunction(goalState))
              {
                  success = true;
                  path = nl;
              }
              nl.add(ns);

              if(nl.size()-1>dr) dr=nl.size()-1;

              if(dr > maxDept) deepCond = true;

                  if(dept)
                      queue.addFirst(nl); // Si es en profundidad
                  else
                      queue.addLast(nl); // Si es en anchura agreg
                  //System.out.println("Agregé un nuevo camino con
              }
              //else System.out.println("Un nodo repetido descarta
      }
      if(validStates==0) deadEnds++;  // Un callejón sin salid
      contador++;
      System.out.println("while: "+contador);
}

if(success) // Si hubo éxito
{
    long elapsed = System.currentTimeMillis()-startTime;
    if(dept) this.setTitle("8-Puzzle (Deep-First Search)");
    else this.setTitle("8-Puzzle (Breadth-First Search)");
    JOptionPane.showMessageDialog(rootPane, "Success!! \nPatl
                                            "Good News!!!",
    System.out.println("Success!");
    String thePath="";
    int n=0;
    int i=startState.getI();
    int j=startState.getJ();
    for(State st: path)
    {
      st.show();
      if(n>0)
        thePath = thePath+st.getMovement();
      n++;
```

muestraEstados: Muestra información sobre el estado actual durante la búsqueda.

```
private void muestraEstados(ArrayList<State> ruta)
{
    System.out.println("======");
    for(State s: ruta)
        s.show();
    System.out.println("======");

}
```

repetido: Comprueba si un estado ya se ha visitado para evitar bucles infinitos.

```
public boolean repetido(ArrayList<State> l, State s)
{
    boolean exist = false;
    for(State ns: l)
    {
        if(ns.isEqual(s)) // Un método propio para c
        {
            exist = true;
            break;
        }
    }
    return exist;
}
```

State

Esta clase representa un estado del rompecabezas y contiene métodos relacionados con la manipulación de los estados, la generación de estados sucesores y la verificación del estado objetivo. State: El constructor inicializa un estado a partir de una cadena que representa la disposición actual del rompecabezas.

```
public State(String sts)
{
    int n=0;
    this.board = new int[4][4];
    for(int i=0;i<4;i++){
        for(int j=0;j<4;j++)
        {
            char c=sts.charAt(n);
            if(c == 'A') board[i][j]=10;
            if(c == 'B') board[i][j]=11;
            if(c == 'C') board[i][j]=12;
            if(c == 'D') board[i][j]=13;
            if(c == 'E') board[i][j]=14;
            if(c == 'F') board[i][j]=15;
            else
                board[i][j]=Character.getNumericValue(c);

            if(board[i][j]==0)
            {
                posI = i;
                posJ = j;
            }
            n++;
            System.out.print(board[i][j]);
        }
    }
```

show: Muestra el estado del rompecabezas en la consola.

```
public void show()
{
    switch(movement)
    {
        case 'u' -> System.out.println("Move UP");
        case 'd' -> System.out.println("Move DOWN");
        case 'l' -> System.out.println("Move LEFT");
        case 'r' -> System.out.println("Move RIGHT")
        case 'n' -> System.out.println("START");
    }
    System.out.println("+--+--+--+--+");
    for(int i=0;i<4;i++)
    {   System.out.print("|");
        for(int j=0;j<4;j++)
        {
            if(board[i][j]>9)
                System.out.print(board[i][j]+"|");
            else
                System.out.print(board[i][j]+" |");
        }
        System.out.println("\n+--+--+--+--+");
    }
```

swap: Intercambia dos fichas en el estado.

```
public void swap(int i,int j)
{
    int aux=board[i][j];
    board[i][j]=0;
    board[posI][posJ]=aux;
    posI=i;
    posJ=j;
}
```

nextStates: Genera estados sucesores posibles a partir del estado actual.

```
public ArrayList<State> nextStates() {
    ArrayList<State> next = new ArrayList<>();
    int[][] newBoard = new int[4][4];

    // Clone board
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            newBoard[i][j] = this.board[i][j];
            if (board[i][j] == 0) {
                posI = i;
                posJ = j;
            }
        }
    }
```

getBoard: Obtiene el tablero de juego actual.

```
public int[][] getBoard()
{
    return board;
}
```

goalFunction: Verifica si el estado actual es igual al estado objetivo.

```
public boolean goalFunction(State goal) {
    boolean success = true;
    int[][] goalBoard = goal.getBoard();
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (goalBoard[i][j] != board[i][j])
                success = false;
                break;
            }
        }
        if (!success) {
            break;
        }
    }
    return success;
```

getI y getJ: Obtiene las coordenadas de la ficha vacía en el tablero.

```
public int getI()
{
    return posI;
}
```

```
public int getJ()
{
    return posJ;
}
```

getMovement: Obtiene el último movimiento realizado en el estado.

```
public char getMovement()
{
    return movement;
}
```

isEqual: Compara dos estados para ver si son iguales.

```
public boolean isEqual(State s) {
    boolean isEqual = true;
    int[][] sBoard = s.getBoard();
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (sBoard[i][j] != board[i][j])
                isEqual = false;
                break;
            }
        }
        if (!isEqual) {
            break;
        }
    }
    return isEqual;
```

Executor

Esta clase se utiliza para ejecutar la animación de los movimientos del rompecabezas después de encontrar la solución. Se encarga de mover las fichas del rompecabezas de acuerdo con la solución encontrada.

```
public void run()
{
    for(int n=0;n<path.length();n++)
    {
            int newI=i;
            int newJ=j;
            char m=path.charAt(n);
            switch(m)
            {
                case 'u' -> newI--;
                case 'd' -> newI++;
                case 'l' -> newJ--;
                case 'r' -> newJ++;
            }
            try
            {
                Thread.sleep((int) (1000/(path.length()-1)));
            }
            catch(Exception ex){ ex.printStackTrace();}
            Icon sw = jBoard[newI][newJ].getIcon();

            jBoard[newI][newJ].setIcon(null);
            jBoard[i][j].setIcon(sw);
            i=newI;
            j=newJ;

        }
        jBoard[i][j].setIcon(new ImageIcon(empty));
```

Comparación del desempeño de los algoritmos

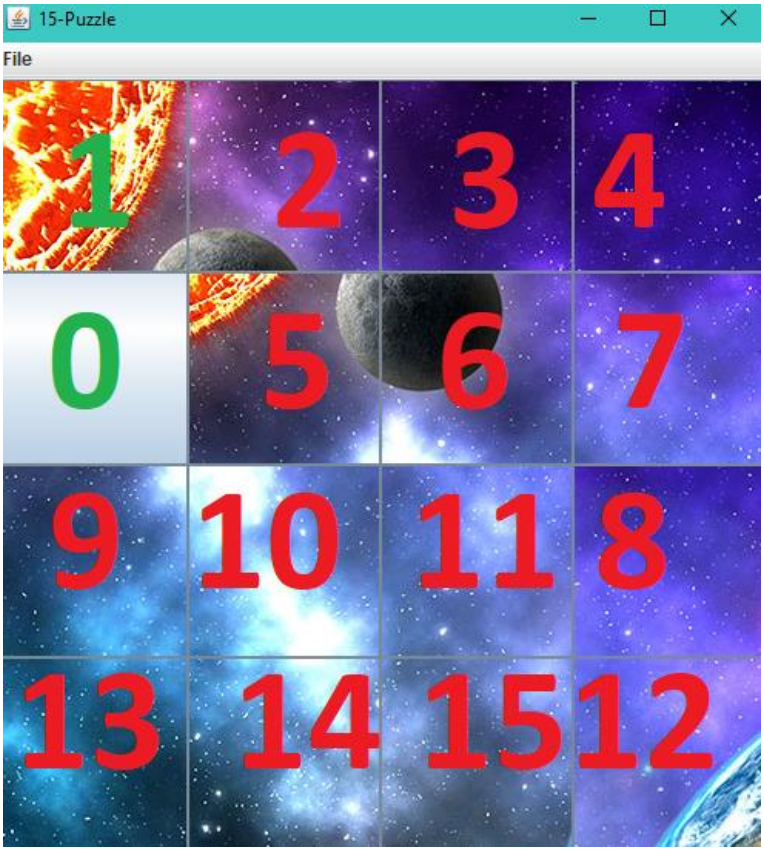
Caso de estudio

Se ha establecido una configuración inicial de las piezas del puzle con la variable “start” de la clase Tablero. Esta configuración se resolverá con Breadth First Search y Depth First Search.

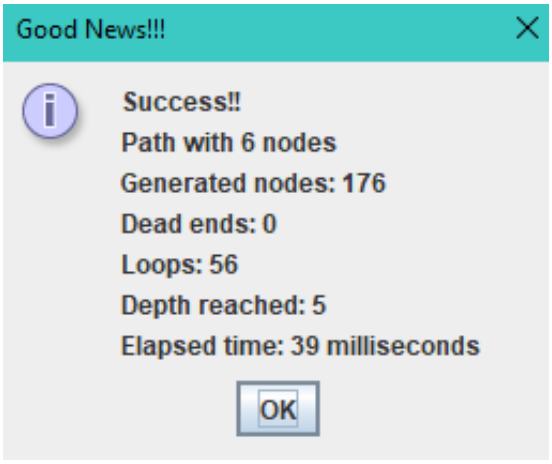
Configuración start en el código:

```
private final String start = "123405679AB8DEFC";
```

Configuración inicial gráficamente:

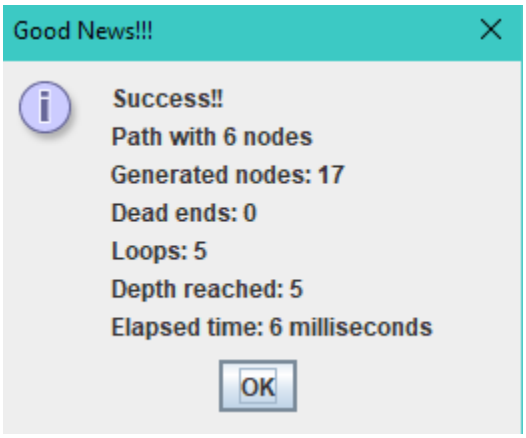


Breadth First Search



Resultado:

Depth First Search



Resultado:

La configuración "123405679AB8DEFC", es un caso interesante para analizar por qué en esta ocasión la Búsqueda en Profundidad (DFS) fue más eficiente que la Búsqueda en Anchura (BFS).

Profundizando en el análisis tenemos que:

La BFS tiende a explorar estados más profundos antes de volver atrás. En este caso, la profundidad máxima alcanzada fue de 176 nodos entre todos los niveles.

DFS puede explorar profundamente en un camino antes de explorar otros. En este caso, la profundidad máxima alcanzada fue de 17 nodos entre los niveles explorados.

La configuración inicial fue más favorable para DFS. Ambos algoritmos tienen lógica para evitar estados repetidos y callejones sin salida. Sin embargo, en este caso, DFS encontró caminos sin salida menos complejos y manejo mejor los estados repetidos. Mientras que para BFS la aleatoriedad en la cola (debido a la exploración de varios caminos en el mismo nivel) afecto negativamente la eficiencia.

CONCLUSIONES

El juego del 15-puzzle es un desafío soluble utilizando algoritmos de búsqueda no informada, como la búsqueda primero en anchura (BFS) y la búsqueda primero en profundidad (DFS). Ambos algoritmos ofrecen enfoques diferentes para abordar el problema y tienen sus propias ventajas y desventajas.

BFS es eficiente para encontrar la solución más corta en términos de movimientos, esto es importante cuando se busca una solución óptima en términos de la menor cantidad de movimientos posible. DFS, por otro lado, se sumerge profundamente en un camino antes de retroceder, lo que puede ser más eficiente en términos de memoria y puede encontrar soluciones en un tiempo razonable.

Ambos algoritmos de búsqueda no informada son valiosos en la resolución de problemas de búsqueda y proporcionan una base sólida para abordar una variedad de desafíos. La elección del algoritmo adecuado dependerá de las características específicas del problema (como el tamaño del tablero, la configuración inicial) y de si se prioriza encontrar la solución óptima en términos de movimientos o se busca una solución rápida y eficiente en términos de recursos.

BIBLIOGRAFÍA

Lukegarrigan. (2021, 5 Junio). *What is Breadth-First Search (BFS) - Codeheir*. Codeheir. https://codeheir.com/2021/06/05/what-is-breadth-first-search-bfs/

Alfonso López. (2020, 11 mayo). *Algoritmo primero el mejor (BFS) C++* [Vídeo]. YouTube. https://www.youtube.com/watch?v=MQ8ZKqE4S2c